An analysis of object-based intelligent image processing and retrieval system

Jia Bin Li, Chun Che Fung and Kok Wai Wong
School of Information Technology
Murdoch University
Western Australia
j.li, l.fung, k.wong@murdoch.edu.au

Abstract—**In order to improve the process of analysis and retrieval of images, it is necessary to examine the execution of such program at the lowest level. This paper reports the results obtained from profiling the execution of an object-oriented image processing and analysis program termed ImageJ. Although profiling has been used in software engineering to identify execution bottlenecks, to our knowledge, it has not been considered as a means of analysis object based distributed image processing systems. The paper summaries the characteristics of the program in four aspects: classes and method codes profiling, method calls, parameter types and return types and CPU percentage. The profiling results show that the total amount of classes that was loaded at runtime is invariant to the change of the image size. Similar behaviours are observed for parameter types and return types. Based on the information, an intelligent monitor can be used to carry out automatic load scheduling and balancing.**

Index Terms—**software profiling; image processing; object oriented programming models; object based distributed systems**

I. Introduction

Parallelism in image synthesis, analysis, processing and retrieval has been widely explored in a spectrum of disciplines. In applications of compression and decompression, every image will be performed by the same set of operations. Thus, specialised digital signal processors (DSP) are pipelined to process continuous data strings to improve the throughput of image processing. Such structure is so-called MISD (multiple instructions single data stream) architecture. On the other hand, an *N*-array processors machine, which is an example of SIMD (single instruction multiple data streams), can manipulate *N* number of images simultaneously. There are a number of image processing systems utilising both multiple instructions and multiple data streams (MIMD) mechanisms. However, those systems dramatically differ to each other in design and implementation.

Shared memory processors (SMP) systems are generally well understood and widely used. In such architecture, a number of processors can concurrently write to (read from) the global memory. Access policy, however, has to be implemented to ensure data consistency from concurrent write operations to the same memory address. Programming on a SMP machine usually requires the use of synchronised concurrent processes or multiple threads.

Distributed systems are another form of parallel systems. In those systems, processors or processing units that consist of a processor and a memory unit are networked using some form of network topologies such as mesh. In order to program theme efficiently, communicating entities have to be considered. These entities may be a set of communicating sequential processes (CSP), or objects, or tokens. Most distributed systems reply on the CSP model such as MPI-C. To our knowledge, only few distributed systems considered object models or data-flow models.

Study of object based distributed systems is conducted through benchmarking and simulation. Benchmarking is used for analysing characteristics of object oriented programs and generating traces from these programs. Then, trace files are inputted to a software simulator for further examination of the design of such structure. This paper will focus on a discussion of the benchmarking analysis on an object oriented image processing program. The program used in this study is termed ImageJ.

This paper serves two purposes. First, it explores the possibilities and advantages of using objects as a means of concurrency to improve performance of image processing and retrieval systems. Second, the profiling results are considered to be used for optimising efficiency of object oriented distributed programs. The rest of the paper is organised as follows. Section II discusses the object oriented models. Section III presents the profiling methodology and the profiling results are given in Section IV. Section V proposes an alternative implementation of the colour histogram using fine grained objects.

II. Object Oriented Models

It has been a misleading concept that execution of object-oriented (OO) programs is slower. Because some incompatibilities between the OOP model and current hardware structures, a virtual machine has to be applied to overcome such incompatibilities. However, the virtual machine causes overheads involved in execution and therefore execution of OO programs is generally considered to be slower comparing to native codes such as C, when they are running on a conventional machine.

Efficient virtual machines have been developed such as the Sun's Java Virtual Machine (JVM) or IBM's Jikes RVM. These virtual machines have been designed to improve the execution efficiency and garbage collection. Due to the existence of the intermediate layer between the high-level

0-7803-9282-5/05/$20.00 ©2005 IEEE

program and the low level machine, the performance of OO software running on those virtual machines however are not able to match the performance of native codes such as C. One solution to overcome the limitation is to develop direct execution machines for OO software such as Java. A number of hardware designs have already been proposed for efficient execution of Java programs in references [1-2]. While these machines can improve execution performance, they are still largely based on a single processor structure.

In the OOP model, communication between objects is via message passing. An object is inherently parallel and self-synchronised in nature. Therefore, the OOP model is well suited for a distributed hardware structure. Most existing distributed systems rely on the Communicating Sequential Processes (CSP) model. Although both the CSP model and the OOP model have an intrinsic nature of parallelism, they approach the issue in very different ways. The CSP model is based on the concept of independent communicating processes acting in concert. It is necessary to make provision for some forms of synchronization mechanism and the model implies the concept of a global address space. In contrast, the OOP model relies on the simple concept of objects communicating with one another. An object includes a concept of state and methods or code. In such case, the synchronisation is intrinsic and the contexts of memory are distributed. Furthermore, the execution environment for the OOP model is highly dynamic as objects are constantly created and destroyed at runtime.

These variations suggest very different hardware architecture and implementation. The CSP model suggests focus should be on a tightly coupled shared memory system with an efficient synchronisation mechanism. The OOP model in contrast suggests a highly distributed system with a communication structure that provides an efficient object referencing mechanism.

III. Methodology

The studied software package is an open source, image processing and analysis software, which is implemented in Java [3]. Figure 1 shows the profiling procedure. Profiling files were generated by using a default profiler provided by Sun JDK 1.4. These profiling files were inputted to the ProfileAnalyser, a Java program that analyses the profiling files. Statistical analysis reports are produced as an output from the ProfAnalyser. Based on these reports, four statistical tables are set up and can be used to illustrate different aspects of the characteristics of the benchmarks:

- **Classes and methods profiling**: measures the total number of classes and method codes loaded during execution.
- **Method calls**: measures the number of messages exchanged between objects.
- **Parameter types and return types**: gives statistical information on different data types used as method parameters or return values.
- **CPU percentage**: measures the amount of the CPU time spends on a particular method.

Since it is not the aim of this paper to design new algorithms for image processing or analysis, we focused on one computation only, calculating colour histogram, to illustrate our proposal. Different sizes of an image were computed to study the effect of the size of an image on the performance of the computation. The image sizes are, respectively, 160x120, 320x240, 480x360 and 640x480. The Image used is shown in Figure 2.

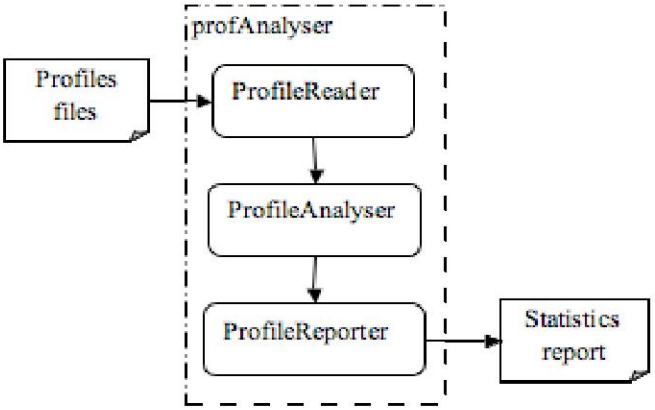

Figure 1: An overview of the Profiling procedure.

Figure 2: A photo of the Nanu flower as the analysis image [4].

IV. Profiling Results

A. Classes and Methods Profiling

Table 1 shows the total number of classes and methods loaded and objects initialised at runtime. In the table, the first row represents different sizes of the profiling image. Row 2-4 shows the total amount of method codes that are loaded during computation regarding various image sizes. Similarly, Row 5-7 shows the total number of classes constructed at runtime. The last row indicates the number of objects created.

Furthermore, the Classes and Methods may be subdivided into two categories: system-supplied and user-defined. System-supplied classes or methods are those included in Sun JDK, which their names generally start with *java*, *javax* or *sun*. The rest are considered as user-defined classes or methods. It is obvious that the percentage of system-supplied

classes or methods is significantly higher than the percentage of user-defined ones for all benchmarks because system-supplied libraries had been well tested and documented and therefore they are generally more efficient and safer to use than those user defined. In addition, using system libraries can help to reduce the complexity of a program.

It is noted that the same measures are almost invariant across different image sizes. This implies that the memory size of the execution machine, i.e. JRE 1.4, was not affected by the size of the input image as the image is an external static entity.

Table 1: statistical information on runtime classes and methods.

Size	160x120	320x240	480x360	640x480
Total Methods	4562	4586	4586	4562
System Methods	4161	4168	4168	4156
User Methods	401	418	418	406
Total Classes	851	851	851	851
System Classes	797	797	797	797
User Classes	54	54	54	54
Initialised Objects	2358	2362	2363	2353

B. *Method Calls – Message Passing between Objects*

Table 2 illustrates the statistical summary of various types of method calls. This differs from the methods described previously. Method codes are functions definitions while method calls are the process to invoke a particular method of an object by another object. Similar to classes and method codes, there are two types of method calls; system calls and user calls. System calls are those calls to system-supplied codes where user calls are calls which invoke user defined codes. Further, a callee method and a caller method may be in the same class – inner calls; otherwise, the call is an external call.

Table 2 shows that the total amount of method calls is influenced by various image sizes. Figure 3 indicates that the total method calls is linearly increasing with the size of the image. In particular, methods such as *getRed*, *getBlue* and *getGreen* got more accesses, as an obvious result, when it operates on a larger image. Later, we will present an object model that is able to parallelise the sequential accesses to these methods.

Table 2: statistical summary of various types of method calls.

Size	160x120	320x240	480x360	640x480
Total Calls	1054491	1584892	2425624	3370895
Calls to system code	1036878	1566622	2407178	3352953
Calls to user code	17613	18270	18446	17942
Inner Calls	385762	806257	1497567	2407492
External Calls	668729	778635	928057	963403

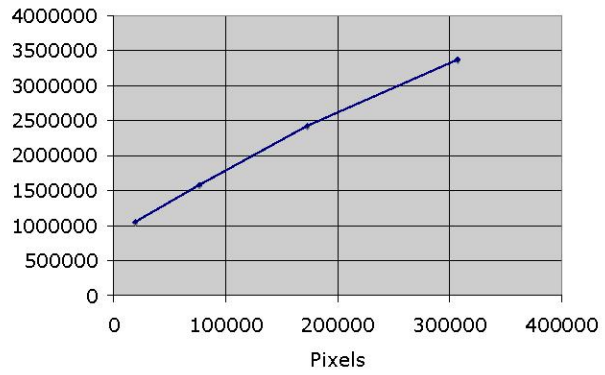

Figure 3: Linear relationship between the total method calls and the size of an image.

C. *Parameter types and return types*

The ProfAnalyser utility program provides statistical information on the parameter types (Table 3) and return types (Table 4) monitored by the profiling program. It is interesting that the profiling results are roughly the same for all the images. However, the percentage of different data types used in parameter and return is not identical as shown in Figure 4 and Figure 5. In particular, object types are frequently used, which are implemented using reference pointers. Consequently, it is important to design an efficient reference handling mechanism to improve the performance of an execution environment. The next mostly used data types are integer and boolean, respectively. For the sake of high performance, operations on integer and boolean have to be optimised due to their regular usage.

Table 3: Statistical information on parameter types across various image sizes.

Size	160x120	320x240	480x360	640x480
Bool	261	261	261	259
Byte	11	11	11	11
Char	35	35	35	35
Short	36	36	36	36
Int	1423	1443	1443	1414
Long	100	100	100	100
Float	54	54	54	54
Double	50	50	50	50
Array	232	232	232	232
Object	2672	2680	2680	2677
Total	4874	4902	4902	4868

Table 4: Statistical information on return types across various image sizes.

Size	160x120	320x240	480x360	640x480
Bool	450	455	455	448
Byte	5	5	5	5
Char	18	18	18	18
Short	19	19	19	19
Int	373	378	378	375
Long	38	38	38	38
Float	13	13	13	13
Double	29	29	29	29
Array	102	103	103	102
Object	1201	1204	1204	1197
Total	2248	2262	2262	2244

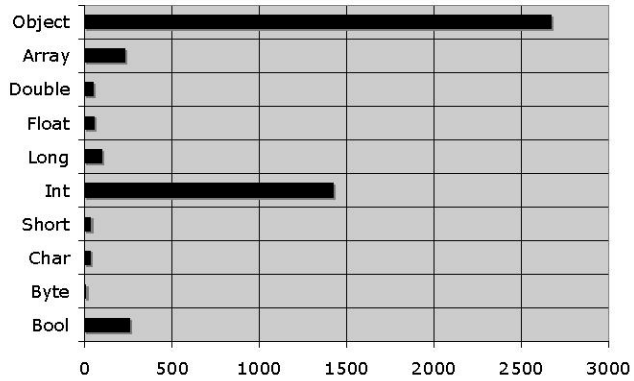

Figure 4: Percentage of parameter types for analysis of the 160x120 image.

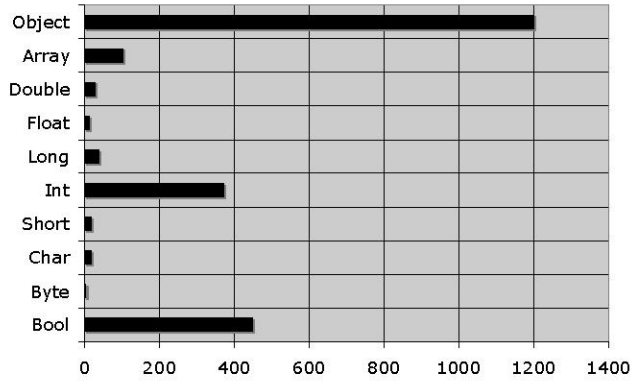

Figure 5: Percentage of return types for analysis of the 160x120 image.

D. *Measurement of CPU time*

Figure 6 shows the screen capture of the execution environment that was performing the histogram operations. It is noted that there were a lot of methods involved in the process, including updating the GUI. However, the method of interest is the highlighted method called *ij.process.ColorProcessor.getHistogram*. The profiler also gives the percentage of the CPU time that dedicated to the particular method. Table 5 shows measurements on CPU time for the *getHistogram* method. However, the measurement for the smallest size image is not available since the cost of the function is too small to sample.

```
Flat profile of 0.73 secs (33 total ticks): Histogram

 Interpreted + native   Method
12.1%    0 +    4    sun.awt.windows.WComponentPeer.pShow
 9.1%    3 +    0    ij.process.ColorProcessor.getHistogram
 6.1%    1 +    1    java.lang.ClassLoader.findBootstrapClass
 3.0%    0 +    1    sun.font.FileFont.getGlyphImage
 3.0%    0 +    1    sun.awt.windows.WWindowPeer.reshapeFrame
 3.0%    0 +    1    sun.awt.windows.WComponentPeer.reshape
 3.0%    1 +    0    java.lang.ClassLoader.defineClass1
 3.0%    1 +    0    java.awt.Window.setClientSize
 3.0%    0 +    1    java.util.zip.Inflater.inflateBytes
 3.0%    0 +    1    sun.awt.windows.WFramePeer.createAwtFrame
 3.0%    0 +    1    ij.ImagePlus.getStatistics
 3.0%    1 +    0    sun.awt.AWTAutoShutdown.notifyThreadBusy
 3.0%    1 +    0    sun.awt.image.ByteInterleavedRaster.verify
 3.0%    1 +    0    java.awt.Container.validate
 3.0%    1 +    0    java.util.zip.InflaterInputStream.read
 3.0%    1 +    0    java.awt.EventQueue.postEventPrivate
 3.0%    1 +    0    ij.process.TypeConverter.convertRGBToByte
 3.0%    1 +    0    ij.plugin.Histogram.run
 3.0%    1 +    0    ij.gui.HistogramWindow.<init>
 3.0%    1 +    0    ij.process.ImageProcessor.getBestIndex
 3.0%    1 +    0    ij.process.ByteBlitter.copyBits
 3.0%    1 +    0    java.awt.image.IndexColorModel.setRGBs
 3.0%    1 +    0    sun.awt.windows.WFramePeer.setIconImage
 3.0%    1 +    0    java.awt.EventQueue.postEvent
90.9%   19 +   11    Total interpreted

  Compiled + native   Method
 3.0%    1 +    0    ij.process.ColorProcessor.getHistogram
 3.0%    0 +    1    ij.gui.NewImage.createByteImage
 6.1%    1 +    1    Total compiled

Thread-local ticks:
 3.0%    1           Unknown: no last frame
```

Figure 6: Screen capture of the profiling of the histogram operation .

Table 5: Measures computation time of four sizes of the image.

CPU (percentage)	Size
9.10%	640x480
6.10%	480x360
3.10%	320x240
N/A	160x120

V. An Object Oriented Implementation of the Colour Histogram Algorithm

Here, we will illustrate how the colour histogram can be implemented by using fine-grained objects to parallelise such execution.

Since the colour histogram algorithm counts of different colour values, the following pseudo-code realises such algorithm:

```
For i = 1 To m
   For j = 1 To n
      color_vector = image.getRGB(i, j);
   End
End
```

Let us assume that the color_vector is an array with length of 256. It is obvious that such algorithm has complexity of O(m·n). Hence, it requires a powerful machine to process very large size images such as satellite photos. In such implementation, the object *image* controls the access to information of the image such as colour values.

However, it is also possible to create a special object called *Pixel*. That is, each object stores information on a single pixel of the whole image. Thus, there will be (m·n) number of objects (pixels) created at start-up. Rather to sequentially access to individual *pixel* object, one can issue a broadcasting message to the group of *pixel* objects. That is, a *pixel* object adds its value to the color_vector that is sent by its neighbours and passes the result array to the next neighbours. This mechanism can be illustrated by Figure 4. In such model, every object needs to receive up to one message from its top or left neighbour while the other message can be ignored, then it passes the results to two other neighbour objects. It is not difficult to see that the complexity of the operation is now determined by passing messages along the longest path (branch), i.e. O(m+n), which is a significant improvement to

the sequential implementation.

There is another parallel implementation of image processing systems using the object model described in [5]. In their model, image features such as HSV are implemented using objects. These objects as primitive objects can be shared by other software components to eliminate the need of recomputation of the same features.

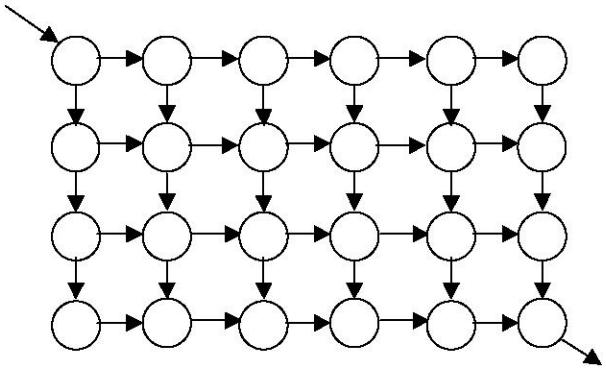

Figure 7: A broadcasting implementation of the colour histogram algorithm.

VI. Conclusions

This paper studied the characteristics of an image processing and analysis software program termed ImageJ by using the profiling technique. Although profiling has been used in software engineering to identify execution bottlenecks, to our knowledge, it has not been considered as a means of analysis object based distributed systems. The paper summaries the characteristics of the program in four aspects: classes and method codes profiling, method calls, parameter types and return types and CPU percentage. The profiling results show that the total amount of classes that was loaded at runtime is invariant to the change of the image size. Similar behaviours are observed for parameter types and return types.

However, increasing in image size leads to increasing the number of messages exchanged between objects. In particular, it has significant impact on the methods that are related to the histogram operations such as *getRed*, *getGreen* and *getBlue*. Therefore, we propose an alternative implementation that utilise more fine grained objects. The complexity of the new implementation is O(m+n) comparing to the sequential implementation that requires O(m·n) operations. However, such model is still too abstract that it has not considered overhead involved in communication. Simulation-based analysis on the new implementation, such as how does the network latency affect on the overall performance, will be reported in the near future. It is also expected that the available information on the profile of execution will form input to an intelligent monitoring unit which will carry out automatic load scheduling and balancing based on the information of the objects. This will improve the image analysis and retrieval process.

Acknowledgement

This project is conducted at the Centre for Enterprise collaboration in Innovative Systems with support received from a grant awarded by the 2004 Murdoch University Research Excellence Grant Scheme (REGS). Jia Bin Li is a recipient of the International Postgraduate Research Scholarship (IPRS) at Murdoch University from 2004.

Reference

[1] W. Chu and Y. Li, "An instruction cache architecture for parallel execution of Java threads," Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT'2003., 2003.

[2] C. M. Chung and S. D. Kim, "A dual threaded Java processor for Java multithreading," Proceedings of 1998 International Conference on Parallel and Distributed Systems, Taiwan, 1998.

[3] ImageJ, Research Services Branch, [url] http://rsb.info.nih.gov/ij/ (accessed date: Jun 2005)

[4] Plants of Hawaii, [url] http://www.hawcc.hawaii.edu/laurab/generalbotany/images/Nanu%20flower.jpg (accessed date: Jun 2005).

[5] K. P. Chung, J. B. Li, C. C. Fung, and K. W. Wong, "A parallel architecture for feature extraction in content-based image retrieval system," The 2004 IEEE International Conference on Cybernetics and Intelligent Systems, pp. 468-473, IEEE, Singapore, 1-3 Dec., 2004.